The Bedwyr system for model checking over syntactic expressions

David Baelde[1], Andrew Gacek[2], Dale Miller[1], Gopalan Nadathur[2], and
Alwen Tiu[3]

[1] INRIA & LIX, École Polytechnique
[2] Digital Technology Center and Dept of CS, University of Minnesota
[3] Australian National University and NICTA

1 Overview

Bedwyr is a generalization of logic programming that allows model checking directly on syntactic expressions possibly containing bindings. This system, written in OCaml, is a direct implementation of two recent advances in the theory of proof search. The first is centered on the fact that both finite success and finite failure can be captured in the sequent calculus by incorporating inference rules for *definitions* that allow *fixed points* to be explored. As a result, proof search in such a sequent calculus can capture simple model checking problems as well as may and must behavior in operational semantics. The second is that higher-order abstract syntax is directly supported using term-level λ-binders and the ∇ quantifier. These features allow reasoning directly on expressions containing bound variables.

2 Foundations

The logical foundation of Bedwyr is the logic called LINC [12], an acronym for "lambda, induction, nabla, and co-induction" that is an enumeration of its major components. LINC extends intuitionistic logic in two directions.

Fixed points via definitions. Clauses such as $A \stackrel{\triangle}{=} B$ are used to provide (mutually) recursive definitions of atoms. Once a set $\mathcal{D}$ of such definition clauses has been fixed, LINC provides inference rules for introducing atomic formulas based on the idea of unfolding definitions. Unfolding on the right of the sequent arrow is specified by the following *definition-right* rule:

$$\frac{\Sigma : \Gamma \vdash B\theta}{\Sigma : \Gamma \vdash A} \text{, provided } A' \stackrel{\triangle}{=} B \in \mathcal{D} \text{ and } A'\theta = A.$$

This rule resembles backchaining in more conventional logic programming languages. The *definition-left* rule is a case analysis justified by a closed-world reading of a definition.

$$\frac{\{\Sigma\theta : \Gamma\theta, B\theta \vdash G\theta \mid A' \stackrel{\triangle}{=} B \in \mathcal{D} \text{ and } \theta \in csu(A, A')\}}{\Sigma : \Gamma, A \vdash G}$$

Notice that this rule uses unification: the eigenvariables of the sequent (stored in the signature Σ) are instantiated by θ, which is a member of a complete set of unifiers (csu) for atoms A and A'. Bedwyr implements a subset of this rule that is restricted to *higher-order pattern unification* and, hence, to a case where *csu* can be replaced by *mgu*. If an atom on the left fails to unify with the head of any definition, the premise set of this inference rule is empty and, hence, the sequent is proved: thus, a unification *failure* is turned into a proof search *success*.

Notice that this use of definitions as fixed points implies that logic specifications are not treated as part of a *theory* from which conclusions are drawn. Instead, the proof system itself is parametrized by the logic specification. In this way, definitions remain fixed during proof search and the *closed world assumption* can be applied to the logic specification. For earlier references to this approach to fixed points see [3, 11, 4].

Nabla quantification. Bedwyr supports the λ-*tree syntax* [6] approach to higher-order abstract syntax [9] by implementing a logic that provides (i) terms that may contain λ-bindings, (ii) variables that can range over such terms, and (iii) equality (and unification) that follows the rules of λ-conversion. Bedwyr shares these attributes with systems such as λProlog. However, it additionally includes the ∇-quantifier that is needed to fully exploit the closed-world aspects of LINC. This quantifier can be read informally as "for a new variable" and is accommodated easily within the sequent calculus with the introduction of a new kind of local context scoped over formulas. We refer the reader to [7] for more details. We point out here, however, that ∇ can always be given minimal scope by using the equivalences $\nabla x.(Ax * Bx) \equiv (\nabla x.Ax) * (\nabla x.Bx)$ where $*$ may be $\supset$, $\wedge$ or $\vee$ and the fact that ∇ is self-dual: $\nabla x.\neg Bx \equiv \neg \nabla x.Bx$. When ∇ is moved under $\forall$ and $\exists$, it *raises* the type of the quantified variable: in particular, in the equivalences $\nabla x \forall y.Fxy \equiv \forall h \nabla x.Fx(hx)$ and $\nabla x \exists y.Fxy \equiv \exists h \nabla x.Fx(hx)$, the variable y is replaced with a functional variable h. Finally, when ∇ is scoped over equations, the equivalence $\nabla x(Tx = Sx) \equiv (\lambda x.Tx) = (\lambda x.Sx)$ allows it to be completely removed. As a result, no fundamentally new ideas are needed to implement ∇ in a framework where λ-term equality is supported.

3   Architecture

Bedwyr implements a fragment of LINC that is large enough to permit interesting applications of fixed points and ∇. In this fragment, *all* the left rules are invertible. Consequently, we can use a simple proof strategy that alternates between left and right-rules, with the left-rules taking precedence over the right rules.

Two provers. The fragment of LINC implemented in Bedwyr is given by the following grammar:

$L0 ::= \top \mid A \mid L0 \wedge L0 \mid L0 \vee L0 \mid \nabla x.\ L0 \mid \exists x.\ L0$

$L1 ::= \top \mid A \mid L1 \wedge L1 \mid L1 \vee L1 \mid \nabla x.\ L1 \mid \exists x.\ L1 \mid \forall x.\ L1 \mid L0 \supset L1$

The formulas in this fragment are divided into *level-0* formulas, given by $L0$ above, and *level-1* formulas, given by $L1$. Implicit in the above grammar is the partition of atoms into level-0 atoms and level-1 atoms. Restrictions apply to goal formulas and definitions: goal formulas can be level-0 or level-1 formulas, and in a definition $A \overset{\triangle}{=} B$, A and B can be level-0 or level-1 formulas, provided that the level of A is greater than or equal to the level of B.

Level-0 formulas are essentially a subset of goal formulas in λProlog (with ∇ replacing $\forall$). Proof search for a defined atom of level-0 is thus the same as in λProlog (and Bedwyr implements that fragment following the basic ideas described in [2]). We can think of a level-0 definition, say, $p\,x \overset{\triangle}{=} B\,x$, as defining a set of elements x satisfying $B\,x$. A successful proof search for $p\,t$ means that t is in the set characterized by B. A level-1 statement like $\forall x. p\,x \supset R\,x$ would then mean that R holds for all elements of the set characterized by p. That is, this statement captures the enumeration of a *model* of p and its verification can be seen as a form of model checking. To reflect this operational reading of level-1 implications, the proof search engine of Bedwyr uses two subprovers: the Level-0 prover (a simplified λProlog engine), and the Level-1 prover. The latter is a usual depth-first goal-directed prover but with a novel treatment of implication. When the Level-1 prover reaches the implication $A \supset B$, it calls the Level-0 prover on A and gets in return a stream of answer substitutions: the Level-1 prover then checks that, for every substitution θ in that stream, $B\theta$ holds. In particular, if Level-0 finitely fails with A, the implication is proved.

As with most depth-first implementations of proof search, Bedwyr suffers from some aspects of incompleteness: for example, the prover can easily loop during a search although different choices of goal or clause ordering can lead to a proof, and certain kinds of unification problems should be delayed instead of attempted eagerly. For a more detailed account on the incompleteness issues, we refer the reader to [14]. Bedwyr does not currently implement static checking of types and the stratification of definitions (which is required in the cut-elimination proof for LINC). This allows us to experiment with a wider range of examples than those allowed by LINC.

Higher-order pattern unification. We adapt the treatment of higher-order pattern unification due to Nadathur and Linnell [8]. This implementation uses the *suspension calculus* representation of λ-terms. We avoid explicit raising, which is expensive, by representing ∇-bound variables by indices and associating a *global* and a *local* level annotation with other quantified variables. The global level replaces raising over existential and universal variables. The local level replaces raising over ∇-bound variables. For example, the scoping in $\forall x.\exists y.\nabla n.\forall z.Fxynz$ is represented by the following annotation: $Fx^{0,0}Y^{1,0}\#_0 z^{2,1}$ (we use lowercase letters for universal variables, uppercase for existentials, the index $\#_n$ for the n-th ∇-bound variable, and write in superscript the annotation $(global, local)$). Using this annotation scheme, the scoping aspects of ∇ quantifiers are reflected into new conditions on local levels but the overall structure of the higher-order pattern unification problem and its mgu properties are preserved.

Tabling. We introduced tabling in Bedwyr to cut-down exponential blowups caused by redundant computations and to detect loops during proof-search. The first optimization is critical for applications such as weak bisimulation checking. The second one proves useful when exploring reachability in a cyclic graph.

Tabling is currently used in Bedwyr to experiment with proof search for inductive and co-inductive predicates. A loop over an inductive predicate that would otherwise cause a divergence can be categorized using tabling as a failure. Similarly, in the co-inductive case, loops yield success. This interpretation of loops as failure or success is not part of the meta-theory of LINC. Its soundness is currently conjectured, although we do not see any inconsistency of this interpretation on the numerous examples that we tried.

Inductive proof-search with tabling is implemented effectively in provers like XSB [10] using, for example, suspensions. The implementation of tables in Bedwyr fits simply in the initial design of the prover but is much weaker. We only table a goal in Level-1 when it does not have free occurrences of variables introduced by an existential quantifier; and in Level-0 when it does not have any free variable occurrence. Nevertheless, this implementation of tabling has proved useful in several cases, ranging from graph examples to bisimulation.

4   Examples

We give here a brief description of the range of applications of Bedwyr. We refer the reader to `http://slimmer.gforge.inria.fr/bedwyr` and the user manual for Bedwyr [1] for more details about these and other examples.

Finite failure. Let *false* be an atom that has no definition. Negation of a level-0 formula G can then be written as the level-1 formula $G \supset false$ and this negation is provable in the level-1 prover if all attempts to prove G in the level-0 prover fail. For example, the formula $\forall y[\lambda x.x = \lambda x.y \supset false]$ is a theorem: *i.e.*, the identity abstraction is always different from a constant-valued abstraction.

Model-checking. If the two predicates P and Q are defined using Horn clauses, then the Level-1 prover is capable of attempting a proof of $\forall x.\ P\ x \supset Q\ x$. This covers most (un)reachability checks common in model-checking. Related examples in the Bedwyr distribution include the verification of a 3 bits addition circuit and graph cyclicity checks.

Games and strategies. Assuming that a transition in a game from position P to position P' can be described by a level-0 formula *step* $P\ P'$ then proving the level-1 atom *win* P defined by

$$win\ P \overset{\triangle}{=} \forall P'.\ step\ P\ P' \supset \exists P''.\ step\ P'\ P'' \wedge win\ P''$$

will determine if there is a winning strategy from position P. If all *win*-atoms are tabled during proof search, the resulting table contains an actual winning strategy.

Simulation in process calculi. If the level-0 atom $P \overset{A}{\longrightarrow} Q$ specifies a one-step transition (process P does an A-action and results in process Q), then simulation

can be written in Bedwyr as follows [5].

$$sim\ P\ Q \stackrel{\triangle}{=} \forall A \forall P'.\ P \xrightarrow{A} P' \supset \exists Q'.\ Q \xrightarrow{A} Q' \wedge sim\ P'\ Q'$$

In dealing with the π-calculus, where bindings can occur within one-step transitions, there are two additional transitions that need to be encoded: in particular, $P \xrightarrow{\downarrow X} P'$ and $P \xrightarrow{\uparrow X} P'$, for bound input and bound output transitions on channel X. In both of these cases, P is a process but P' is a name abstraction over a process. The full specification of (late, open) simulation for the π-calculus can be written using the following [7].

$$
\begin{aligned}
sim\ P\ Q \stackrel{\triangle}{=}\ & [\forall A \forall P'.\ P \xrightarrow{A} P' \supset \exists Q'.\ Q \xrightarrow{A} Q' \wedge sim\ P'\ Q'] \wedge \\
& [\forall X \forall P'.\ P \xrightarrow{\downarrow X} P' \supset \exists Q'.\ Q \xrightarrow{\downarrow X} Q' \wedge \forall w. sim\ (P'w)\ (Q'w)] \wedge \\
& [\forall X \forall P'.\ P \xrightarrow{\uparrow X} P' \supset \exists Q'.\ Q \xrightarrow{\uparrow X} Q' \wedge \nabla w. sim\ (P'w)\ (Q'w)]
\end{aligned}
$$

Notice that the abstracted continuation resulting from bound input and bound output actions are treated by the $\forall$-quantifier and the ∇-quantifier, respectively. In a similar way, modal logics for the π-calculus can be captured [13]. If *sim*-atoms are tabled during proof search, the resulting table contains an actual simulation. Bisimulation is easily captured by simply adding the symmetric clauses for all those used to define *sim*.

Meta-level reasoning. Because Bedwyr uses the ∇ quantifier and the λ-tree approach to encoding syntax, it is possible to specify provability in an object logic and to reason to some extent about what is and is not provable. Consider the tiny fragment of intuitionistic logic with the universal quantifier $\overline{\forall}$ and the implication $\Rightarrow$ in which we only allow atoms to the left of implications. If the formula $\overline{\forall}x.\ (p\ x\ r \Rightarrow \overline{\forall}y.\ (p\ y\ s \Rightarrow p\ x\ t))$ is provable in this logic then one would expect r and t to be syntactically equal terms. In searching for a proof of this formula, the quantified variables are replaced by distinct eigenvariables: therefore, the only way the formula could have been proved is for $p\ x\ t$ to match $p\ x\ r$, hence $r = t$. Provability of a formula B from a list of atomic formulas L can be specified by the following meta-level (Bedwyr-level) judgment $pv\ L\ B$:

$$pv\ L\ B \stackrel{\triangle}{=} memb\ B\ L. \qquad\qquad pv\ L\ (\overline{\forall}B) \stackrel{\triangle}{=} \nabla x.\ pv\ L\ (B\ x).$$
$$pv\ L\ (A \Rightarrow B) \stackrel{\triangle}{=} pv\ (A :: L)\ B.$$

Here, *memb* and :: are the usual predicate for list membership and the non-empty list constructor. Object-level eigenvariables are specified using the meta-level ∇-quantifier. The above observation about object-logic provability can now be stated in the meta-logic as the following formula, which is provable in Bedwyr:

$$\forall r \forall s \forall t.\ pv\ nil\ (\overline{\forall}x.\ (p\ x\ r \Rightarrow \overline{\forall}y.\ (p\ y\ s \Rightarrow p\ x\ t))) \supset r = t.$$

5 Future Work

We are working on several improvements to Bedwyr, including more sophisticated tabling and allowing the suspension of goals containing non-higher-order-

pattern unification (rescheduling them when instantiations change them into higher-order pattern goals). We will also explore using tables as proof certificates: for example, when proving that two processes are bisimilar, the table stores an actual bisimulation, the existence of which proves the bisimilarity. Bedwyr is an open source project: more details about it can be found at `http://slimmer.gforge.inria.fr/bedwyr/`.

References

1. David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. *A User Guide to Bedwyr*, November 2006.
2. Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325. MIT Press, 1991.
3. Jean-Yves Girard. A fixpoint theorem in linear logic. An email posting to the mailing list linear@cs.stanford.edu, February 1992.
4. Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In *Proc. LICS 1997*, pp. 434–445, IEEE Comp. Soc. Press, 1997.
5. Raymond McDowell, Dale Miller, and Catuscia Palamidessi. Encoding transition systems in sequent calculus. *Theoretical Computer Science*, 294(3):411–437, 2003.
6. Dale Miller. Abstract syntax for variable binders: An overview. In John Lloyd and et. al., editors, *Computational Logic - CL 2000*, number 1861 in LNAI, pages 239–253. Springer, 2000.
7. Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, October 2005.
8. Gopalan Nadathur and Natalie Linnell. Practical higher-order pattern unification with on-the-fly raising. In *ICLP 2005: 21st International Logic Programming Conference*, volume 3668 of *LNCS*, pages 371–386, Sitges, Spain, 2005. Springer.
9. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.
10. Konstantinos Sagonas, Terrance Swift, David S. Warren, Juliana Freire, Prasad Rao, Baoqiu Cui, Ernie Johnson, Luis de Castro, Rui F. Marques, Steve Dawson, and Michael Kifer. *The XSB Version 3.0 Volume 1: Programmer's Manual*, 2006.
11. Peter Schroeder-Heister. Rules of definitional reflection. In *Proc. LICS 1993*, pages 222–232. IEEE Comp. Soc. Press, 1993.
12. Alwen Tiu. *A Logical Framework for Reasoning about Logical Specifications*. PhD thesis, Pennsylvania State University, May 2004.
13. Alwen Tiu. Model checking for π-calculus using proof search. In M. Abadi and L. de Alfaro, editors, *CONCUR*, volume 3653 of *LNCS*, pages 36–50. Springer, 2005.
14. Alwen Tiu, Gopalan Nadathur, and Dale Miller. Mixing finite success and finite failure in an automated prover. In *Proc. of ESHOL'05: Empirically Successful Automated Reasoning in Higher-Order Logics*, pages 79 – 98, December 2005.